A Generic Programming Concept Web

David R. Musser

Incomplete Draft, December 21, 1999

Contents

1 Introduction

This hypermedia document is an attempt to define and connect some of the main concepts of *generic programming*, a relatively new approach to computer programming. It is also intended as an illustration of computer-supported concept development and knowledge representation. Thus the document is written as interlinked text pages, which, together with WWW browser capabilities, put the concepts in a more accessible (and malleable) form than a traditional linear presentation could do. The document is an example of what I call a *concept web*, in which the main content of Web pages is definitions and discussions of the concepts of some particular subject area, and the links between the pages include ones that show concept refinement relations. In the print version of the document, hypertext links are replaced by page references.

2 Concept Webs

2.1 Overview and examples

Concept webs are hypermedia documents containing definitions and discussions of the key concepts of a subject, internally linked according to conceptual structure and also linked to examples and to other related material, including such things as animations, simulations, experiments, interactive database searches, etc. The potential uses of concept webs are many, in education, research, and commercial enterprise. The World Wide Web currently provides an excellent medium for dissemination of concept webs; cooperative development of concept webs is also supported reasonably well.

One large existing Web site that shows a lot of the potential of concept webs (though its author doesn't use that term) is SGI's Standard Template Library (STL) documentation, which makes extensive use of concept definitions and refinement in documenting the requirements and properties of a set of standard algorithms and data structures. See

http://www.cs.rpi.edu/~musser/STL_concept_web, where I've added a graphical representation of a major part of SGI's pages, with links into the pages.

The STL documentation example is an excellent but imperfect example of concept webs as I currently see them being developed as a means of informating academic and commercial disciplines.[1] Indeed, we should probably recognize that concept webs will never be perfect; they can always be revised and extended, but in the educational context this is a *virtue*, not a drawback. For example, the STL Web pages are static, but there are lots of possibilities for making them active (and interactive). The Web pages for particular algorithms in the STL Web pages could for example contain animations of the algorithms. They don't currently, but one can see good examples of such animations in the separate SGI Java Algorithm Demo site. Developing such active content is an area where students could be heavily involved.

The SGI example and the example of generic programming concepts developed in this document are heavily computer science-oriented, but the idea of hypermedia concept descriptions and refinement hierarchies can be applied to any subject area.

2.2 The Concept of a Concept

What is a *concept*? There are many definitions, but as I will use the term here, it means

- a set R of requirements (§2.5), and
- a set A of abstractions (§2.3),

such that an abstraction is included in A if and only if it satisfies all of the requirements in R. Though this is a technical definition, I believe many everyday uses of the term "concept" can be derived from it.

The requirements of a concept are explicitly stated in textual form in what is called a *concept description*. It is certainly conceivable that concept descriptions could be stated in a completely formal mathematical language, so that discussions of concepts could be carried out with a very high degree of mathematical rigor. It is also conceivable that processing of this language could be automated to a great degree using algorithmic decision procedures or heuristic artificial intelligence approaches. Work in this direction and further references can be found in [1, 2, 4]. Given present limitations of computer science and computer hardware, the formal approach may be too restrictive for some. In this document I choose to state concepts in informal natural language, but the framework and terminology are based on a fairly well-developed formalism.

This definition of concept and many of the directions in which we would like to develop concept webs are derived mainly from two sources: the field

[1] The word "informating" was coined by Shoshana Zuboff in her 1988 book, *In the Age of the Smart Machine*, to describe the way that information technology could be used to expand knowledge, particularly in the workplace, in contrast to the concerns associated with "automating" of work.

of formal concept analysis [6, 7], and the Tecton concept description language [1, 2, 4]. Formal concept analysis seems heretofore to have been mainly applied to concepts in which the abstractions are merely names or simple descriptions of objects, so that the only real abstraction going on is from objects to concepts, making the types of concept analysis often no more complicated (or interesting) than typical relational database operations. Tecton, on the other hand, was designed for, and has been applied to examples in which the abstractions are quite complex, such as algebraic structures from mathematics and abstract data types from computer science, so that the step from abstractions to concepts is a second-level abstraction step and one that requires nontrivial inferences that are beyond the scope of relational database systems.

2.3 Abstractions

What is an *abstraction*? A dictionary definition of "to abstract" is "to consider apart from application to or association with a particular instance" (Merriam-Webster's Collegiate Dictionary, Tenth Edition). For purposes of this document, I will define an abstraction as a set of labeled sets, where the labels are called *sorts*, and a set of labeled operations on the sets, where the operations have a specified *arity*, which is a list of specified sorts of the operation's inputs and outputs. This is in fact only a mild generalization of the way a *data abstraction* (a.k.a. a *data type* or simply a *type*) is often defined in computer science, but actually the definition is broad enough to capture also a notion of algorithms and other fundamental computer science terms. An algorithm, for example, is an abstraction in which there is only one operation (the algorithm) in the set of operations and the set of objects is empty. I believe that the computer science definition can also be usefully applied to other non-computational domains, but in this document we will not attempt to demonstrate that.

Requirements on abstractions are of two kinds: interfaces and properties.

2.3.1 Interfaces

The interface requirements state the syntax with which the objects of an abstraction are referenced and the operations are invoked. *Sorts* are introduced as names for sets of objects. For operations, interfaces specify the operation name and its *arity*, a list of sorts of its inputs and outputs. [[This should be elaborated.]]

2.3.2 Properties

Properties are semantic requirements and resource (time and space) constraints on the operations of an abstraction. [[This should be elaborated.]]

2.4 Models of Concepts

Another term frequently applied to abstractions and concepts is *model*: an abstraction that belongs to a concept is also said to be a *model of the concept*.

We also sometimes say that the abstraction or model *conforms to the concept*. When discussing a concept, it is convenient to say "models include ...", with the understanding that this means models of the concept under discussion.

2.5 Requirements Specifications

How do we state the requirements that an abstraction must satisfy in order to belong to a concept? We focus here on one way (of many possible ways) of stating such requirements, using natural language. We first consider a *simple concept description* and then expand what concept descriptions can include in several useful ways.

A simple concept description has two parts, a *syntactic part* and a *semantic part*.

- The syntactic part describes the interfaces (§2.3.1) of the objects and operations; that is, in order to belong to the concept an abstraction must have certain objects and operations with certain interfaces.
- The semantic part describes the properties (§2.3.2) that the objects and operations of an abstraction must have in order to belong to the concept.

2.6 Concept Refinement

In general a concept description can have some additional components beyond the syntactic and semantic components of a simple concept description:

Refinement An important component is a list of other concept descriptions that it *refines*. That is, all of the requirements of the listed descriptions are incorporated into this description, which means that in order for an abstraction to belong to the concept it must belong to all of the concepts on the list (and it must also satisfy the requirements stated in the syntactic and semantic components of the present description).

We might also say that the concept description *inherits from* the concept descriptions it refines. However, inheritance has come to have a fairly specialized meaning in computer science, particularly in object oriented programming, so we shall generally avoid the term here, using "refinement" instead.

In concept webs, a concept description should contain hypertext links to each of concept descriptions that it refines.

Use Similarly, a concept description may include a list of (names of or links to) other concept descriptions that it *uses* in stating the description. The difference between use and refinement is that the meaning of the symbols in the concept descriptions on the use list is not changed, whereas symbols in the concept descriptions on the refines list generally have their meanings modified by additional requirements (stated in the present description or coming from other descriptions on the refines lists). We can think of the concepts mentioned on the uses list as being incorporated intact as parts ("sub-assemblies") of the current concept.

We do not (intentionally) permit circular chains of refinement or use. Thus refinement and use relations on concept descriptions are hierarchical; in terms of graph theory, they form an acyclic graph. Concept descriptions may nevertheless contain other links besides refinement and use links, and there is no requirement of noncircularity among the total set of connections. Thus we shall use the term *conceptual hierarchy* when referring to the acyclic graphs of refinement and use, but also speak of the *concept graph* or *concept web* in reference to a set of concept descriptions and all of their connections.

Hierarchical development of concepts through refinement and use is what makes it possible to build up descriptions incrementally, rather than having to state long, flat lists of requirements for complex concepts. It is the main support for categorization and other related conceptualization mechanisms, including generalization (removing requirements—going up the refinement hierarchy) and specialization (adding requirements—going down the refinement hierarchy).

2.7 Knowledge Representation

A knowledge representation is a way of expressing knowledge for communication (possibly human-to-human, human-to-machine, or machine-to-machine). Refinements include computer knowledge representation (§2.7.1).

A knowledge representation can be general (e.g., the Web-page representation of concepts under discussion in and illustrated by this document) or, more commonly, specific to a subject area (e.g., providing special alphabets and notation for expressing mathematical knowledge). Specific subject area knowledge representations might be developed as refinements of a general knowledge representation.

2.7.1 Computer Representation of Knowledge

Computer representation of knowledge (a.k.a. *machine representation of knowledge*) is a knowledge representation (§2.7) that is implemented or implementable in a computer medium, including operations for storing and examining units or collections of represented knowledge, and for searching among them to select certain units or collections. Not all of these operations are necessarily automated in the computer medium.

3 An Example of Concept Webs: Programming Concepts

As the main example of concept development in this document, we develop parts of a concept web of programming concepts, concentrating mainly on generic programming. So far this development contains only textual material; I would like to add a graphical depiction of the conceptual hierarchy, and it ideally should be greatly expanded with many examples and other kinds of related material such as animations, simulations, experiments, interactive database searches, etc.

3.1 Programming

Programming is the process of developing computer programs (§3.3). (There are of course other concepts of programming, such as television programming, and in a larger concept web that included them we would relabel this concept as "computer programming.") Its operations include writing, testing, debugging, and extending programs.

Models of the programming concept include both programming carried out manually by humans (programmers) and automatically by computer programs (program generators such as compilers). The "manual" programming activity of human programmers is often aided by programming tools which themselves are usually computer programs (text-editors, compilers, debugging systems, documentation aids, etc.).

Refinements of this concept include what are often called "programming paradigms," such as *procedural programming*, *functional programming*, *object-oriented programming*, *logic programming*, and *generic programming*. Each of these deserves expansion into a concept web. In this document we concentrate on generic programming (§3.8). The way we define terms is based on classic definitions of algorithms and computational methods such as Knuth's [3], but generalized to encompass generic programming.

3.2 Programming Language

A programming language is a language for expressing computer programs (§3.3).

3.3 Computer Program

A computer program is a definite computational method (§3.5) that is implemented or implementable in a computer medium. It is expressed in a programming language (§3.2).

3.4 Generic Computational Method

A *generic computational method* (a.k.a. an *abstract computational method*) is a method for solving a specific type of problem by means of a finite set of steps operating on *inputs*, which are quantities given to it before execution of the steps begins, and producing one or more *outputs*, which have a specified relation to the inputs. The number of steps in the method is required to be not only finite but also independent of the inputs. The method is also required to be *resource constrained*, which means there are requirements on all operations of all steps of the method that constrain the resources (time, space) that can used in executions of the method.

Execution of steps may repeat other steps, so that although the set of steps is finite, executions of them may produce an infinite sequence of steps—finite termination is not a requirement (it *is* a requirement of the generic algorithm (§3.6) concept). Nonterminating computational methods are useful, such as computer operating systems or event-driven simulation systems. Even though the execution of such methods does not terminate, we are still generally interested in bounding the number of steps taken in producing some partial output (as in proving response-time guarantees for an operating system).

In order to bound the resources—time and space—consumed during an execution of the method, we first need bounds on the resources consumed by individual steps. This motivates the resource-constraint requirement on generic computational methods.

Effectiveness of a computational method is the property that all operations of all steps of the method "must be sufficiently basic that they can in principle be done exactly and in a finite length of time." (Knuth [3] adds "by someone using pencil and paper," but it is a philosophical question whether humans have any computational capability beyond the effectiveness of machines.) As defined here, effectiveness of generic computational methods follows from their resource constraint requirement.

3.5 Definite (Nongeneric) Computational Method

Definiteness of a computational method is the property that each step of the method "must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case" [3]. This includes the property that it must be unambiguous which step, if any, follows the current step in any execution of the method.

Generic computational methods do not usually have the property of definiteness, since they only place certain requirements on the operations used, without necessarily pinning them down. They are defined in terms of concepts (collections of abstractions) rather than in terms of individual abstractions. When the requirements are drawn so tight that there is no ambiguity—each concept involved contains only a single abstraction—a generic computational method has the property of definiteness; we also say that it is *nongeneric*. Alternatively, we might have a concept containing more than one abstraction, but we achieve definiteness by selecting from it a single abstraction.

Note also that resource-constraint requirements place some limitations on just how "indefinite" the steps of a method may be.

3.6 Generic Algorithm—A Finite Generic Computational Method

Finiteness of a computational method is the property that the number of steps in any execution of the method must be finite. The finiteness property is also called *termination*, and the method is said to be *terminating*. *Generic algorithm* is a synonym for *finite generic computational method*, a generic computational method (§3.4) with the additional property of finiteness. Every abstraction that belongs to a generic algorithm concept must have the termination property.

Note that among the abstractions belonging to a generic computational method concept, some might be terminating while others are nonterminating.

3.7 Algorithm—A Finite and Definite Computational Method

An *algorithm* is a definite computational method (§3.5) with the additional property of finiteness (§3.6). Equivalently, an algorithm is a generic algorithm (§3.6) with the additional property of definiteness (§3.5).

Thus an algorithm has inputs and outputs and has all of the properties of definiteness, finiteness, and being resource constrained (§3.4) (which implies effectiveness).

A computational method which is intended to be an algorithm but which sometimes doesn't terminate (i.e., some of its execution sequences are infinite) is often still called an algorithm, but one with a "bug" in it.

3.8 Generic Programming

Generic programming is a form of programming (§3.1) that emphasizes the design and programming of generic computational methods (§3.4) (also called generic programs), usually by abstracting from collections of definite computational methods. Another definition of generic programming is *programming in terms of concepts*, as opposed to programming in terms of individual abstractions.

The value of a generic program is that it can replace a large collection of definite programs and is much easier to maintain. There is a crucial concern with efficiency; we are not interested in generic programs whose specializations are so much less efficient than the original programs in the collection that one would be tempted to keep the original collection in spite of the difficulty of maintaining it.

Most current major programming languages provide no support for generic programming. Of the ones that do, the language in widest use is C++, which has *template* functions and classes that permit direct expression of generic computational methods, though in a limited way. Ada 95 provides a similar level of support with its generic procedures and packages. (Ada 83 was the first major language to provide such support, well before C++ did.)

3.9 Generic Algorithm Specialized by Input Type

This concept is a narrowing of the generic algorithm (§3.6) concept by restrictions on the form of input type. Subconcepts include generic sequence algorithms, generic tree algorithms, generic graph algorithms, generic algebraic algorithms, etc.

3.10 Generic Sequence Algorithm

A *generic sequence algorithm* is a generic algorithm (§3.6) that takes one or more linear sequences as inputs. Thus for example generic divide-and-conquer sequence algorithms (§3.14) are a refinement of the generic divide-and-conquer algorithm (§3.13) concept, in which the inputs are linear sequences (as opposed, say, to sets).

Other important examples of classification of algorithms according to their input types include generic matrix algorithms (§3.19.2), generic tree algorithms (§3.19.3), generic graph algorithms (§3.19.4), etc.

3.11 Generic Algorithm Specialized by I/O-Relation

This concept is a narrowing of the generic algorithm (§3.6) concept by restrictions on the form of input-output relations. Subconcepts include generic sorting algorithms, generic search algorithms, generic graph traversal algorithms, generic polynomial addition algorithms, etc.

3.12 Generic Algorithm Specialized by Strategy

This concept is a narrowing of the generic algorithm (§3.6) concept in terms of strategies used in structuring the steps of the algorithm. Subconcepts include generic divide-and-conquer algorithms (§3.13), generic dynamic programming algorithms (§3.19.5), generic greedy algorithms (§3.19.6), etc.

3.13 Generic Divide-and-Conquer Algorithm

A *generic divide-and-conquer algorithm* is a generic algorithm (§3.6) whose steps are structured according to the following strategy:

1. Construct the output directly and return it, if the input is simple enough. Otherwise:
2. Divide the input into two or more (a finite number) of smaller inputs.
3. Recursively apply the algorithm to each of the smaller inputs produced in the first step.
4. Combine the outputs from the recursive applications to produce the output corresponding to the original input.

This concept is one of many known ways of narrowing the generic algorithm concept in terms of a strategy (§3.12), which gives a specific structure to the steps of the algorithm.

3.14 Generic Divide-and-Conquer Sequence Algorithm

A *generic divide-and-conquer sequence algorithm* is a generic divide-and-conquer algorithm (§3.13) that is also a generic sequence algorithm (§3.10).

Subconcepts might be based on different ways of dividing a sequence up into smaller ones.

3.15 Generic Sorting Algorithm

A *generic sorting algorithm* is a generic sequence algorithm (§3.10) whose input-output relation is specialized (§3.11) to the relation that the input is a finite sequence of values and the output is a permutation of the input and is ordered according to some criterion.

The requirement of "ordered according to some criterion" is an example of the indefiniteness that makes the algorithm generic. We cannot obtain a non-generic algorithm until we specify a particular ordering relation on the values. However, we can (and should, to ensure the algorithm has a generic implementation) put certain requirements on the ordering relation. Usually the requirement is stated that it must be a total order, but the least restrictive requirement is what is called a "strict-weak-order." (These terms should have links into a concept web that defines various concepts of order relations.)

3.16 Generic Search Algorithm

A *generic search algorithm* is a generic algorithm (§3.6) whose input-output relation is specialized (§3.11) to the relation that the inputs are a collection of values and another value, and the output is the boolean value true if the other value belongs to the collection, false otherwise.

Subconcepts include generic linear search (§3.19.1) and generic binary search (§3.18).

3.17 Generic Divide-and-Conquer Sorting Algorithm

A *generic divide-and-conquer sorting algorithm* is a generic sorting algorithm (§3.15) that is also a generic divide-and-conquer sequence algorithm (§3.14). Subconcepts include generic quicksort (§3.19.7) and generic mergesort (§3.19.8).

3.18 Generic Binary Search

A *generic binary search* is a generic search algorithm (§3.16) that is also a generic divide-and-conquer algorithm (§3.13), in which the input is divided into two equal (or equal as possible) parts in the divide step.

3.19 Some Areas for Expansion of this Concept Web

There are many ways this generic programming concept web could be expanded, such as:

3.19.1 Generic linear search

3.19.2 Generic matrix algorithm

3.19.3 Generic tree algorithm

3.19.4 Generic graph algorithm

3.19.5 Generic dynamic programming algorithm

3.19.6 Generic greedy algorithm

3.19.7 Generic quicksort

3.19.8 Generic mergesort

References

[1] Kapur, D., Musser, D. R., and Stepanov, A. A., "Tecton: A Language for Manipulating Generic Objects," *Proc. of Program Specification Workshop*, J. Staunstrup, ed., Aarhus, Denmark, August 1981, chapter in *Lecture Notes in Computer Science*, Vol. 134, Springer-Verlag, Berlin, 1982.

[2] Kapur, D. and Musser, D. R., "Tecton: A Framework for Specifying and Verifying Generic System Components," RPI Computer Science Department Technical Report 92-20, Troy, NY, July 1992.

[3] Knuth, D. E., *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1968.

[4] Musser, David R.,The Tecton Concept Description Language, `http://www.cs.rpi.edu/~musser/gp/tecton/tecton1.ps.gz`, Universität Tübingen, July, 1998.

[5] Musser, David R., Tecton Description of STL Container and Iterator Concepts, `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz`, Universität Tübingen, August, 1998.

[6] Wille, Rudolf. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, Ordered Sets, pages 445–470. Reidel, Dordrecht-Boston, 1982.

[7] Wille, Rudolf. Concept lattices and conceptual knowledge systems. Computers and Mathematics with Applications, 23:493–522, 1992.